Synthesizing safe control-command systems out of reusable components

Salam Hajjar a, Emil Dumitrescu b,*, Laurent Pietrac b, Eric Niel b

a Grenoble INP; Gipsa-Lab (UMR5216), F-38400 Saint Martin d’Hères, France
b Université de Lyon, INSA Lyon, Ampère (UMR5005), F-69621 Villeurbanne, France

A R T I C L E   I N F O

Article history:
Received 18 October 2014
Received in revised form 30 April 2015
Accepted 11 May 2015

Keywords:
Formal verification
Discrete controller synthesis
COTS
Simulation
Embedded systems
Control-command

A B S T R A C T

This paper presents a safe design method for control-command embedded systems. It investigates the problem of building control-command systems out of Commercial off the shelf (COTS) components. The design method proposed uses in synergy the formal verification (FV) and the Discrete Controller Synthesis (DCS) techniques. COTS are formally specified using temporal logic and/or executable observers. New functions are built by assembling COTS together. As the COTS assembly operation is seldom error-free, behavioral incompatibilities may persist between COTS. For these reasons, COTS assemblies need to be formally verified and if errors are found, an automatic correction is attempted using DCS. The control-command code generated by DCS needs hardware specific post-processing: a structural decomposition, followed by a controllability assessment, followed by a dedicated formal verification step, ensuring that no spurious behavior is added by DCS. The resulting system is ready for hardware (e.g. FPGA) implementation.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Due to design constraints bounding delays, costs and engineering resources, component re-usability has become a key issue in embedded systems’ design. The expertise of the design process has been shifted from code writing to the efficient management of Commercial off the Shelf (COTS) libraries: by assembling adequately COTS components new functions can be quickly built.

Paradoxically, the brute force application of this method has led to important design and maintenance costs, far from the theoretically expected gains (Abts, 2002). Indeed, COTS implement complex functions and offer the comfort of the abstraction: designers are allowed to focus on what function a COTS achieves and not worry about how it is achieved. Much more than mere comfort, this idea is the cornerstone in code reuse. Yet this ideal perception is distorted by the COTS providers’ idiosyncrasy which brings back to a more pessimistic reality:

• the way a function is implemented is often important. It often needs to fit specific implementation needs: timing, protocol, performance, etc.
• the way an implementation is assessed is crucial. It is often true, and not always clearly stated, that its correct operation is not simply granted, but conditioned by the way it is used.

This situation depicts an antagonism between the genericity expected for a COTS, and implementation particularities which it often fails to handle correctly.

Hence, by assembling COTS which have been separately designed, the resulting interactions cannot be entirely anticipated. Unwanted global behaviors may occur, although each component taken separately is considered free of errors. This is why ensuring a safe behavior is vital for discovering subtle bugs, which are very difficult to uncover by simulation. Even though this technique has become mature, designers must correct errors manually. This process is extremely tedious; searching for bug-fixes is a tricky task requiring high level expertise and deep insight of the design at hand. At any rate, this task remains error-prone: by attempting to manually correct an error, another error might be introduced, which creates a vicious circle.

These elements account for the difficulty inherent to a safe design flow. The weight of the verification process is difficult to establish precisely, but it is considered to be around 50–70% of the design flow time, according to designers feedback. Bacchini et al. (2005) account for
a dramatic growth of the verification challenge, mentioning a double exponential growth, due to growing design complexity. They advocate component re-use within adequate verification methodologies.

Yet additional challenges are introduced by component re-use. COTS need to be assessed by comparing their documented capabilities to the target requirements. This requires a formalization effort, capturing both interface (Adly & Sitaraman, 1999; De Alfaro & Henzinger, 2001) and behavior (Chung & Cooper, 2004). Formal verification is advocated by Xie, Yang, and Song (2007) and Soliman and Frey (2011), together with compositional reasoning (Dragomir, Ober, & Percebois, 2013; Xie et al., 2007) in order to support the COTS-based design.

Despite the availability of these mature tools and methods, the act of finding a bug-fix remains extremely difficult. Reusable COTS are not intended to be modified, and are sometimes not even readable by humans; no insight is available, except the one provided by the COTS “documentation”. Besides, if COTS reused are individually correct but not suitable for direct assembly, it is a hard task finding out and programming by hand the dynamics establishing a proper interaction between them. In this situation, an automatic generation of a proper interaction between COTSs (Altisen, Clodic, Maraninchi, & Rutten, 2003; Borrione, Morin-Alloy, & Oddos, 2012) acts considerably in reducing the search time for bug-fixes. However, such an approach should provide support for design languages, requirement expression, and for modularity, as can be found in Delaval, Marchand, and Rutten (2010) and Yu, Gamanie, and Rutten (2008) for embedded software design.

Additional care is required when the target design has a hardware implementation, as control-command COTSs are not specifically designed for subsequent use with such code generation techniques, as advocated in Delaval et al. (2010). When COTS components are handled, it is desirable that code generation enforces desired behaviors without altering the COTS interface. This calls for a specific design method. Guillet, De Lamotte, Le Griguer, Rutten, and Diguert (2014) apply UML modeling to system-on-chip reconfigurable design; DCS is used here as a tool for handling dynamic configuration switching on high performance dynamically reconfigurable FPGA systems.

This work advocates the use of the Discrete Controller Synthesis (DCS) technique (Marchand, Bournai, Le Borgne, & Guernic, 2000) in order to generate correct-by-construction code. A design method is proposed which highlights the synergy between usual design techniques, as advocated in Delaval et al. (2010). When COTS components are handled, it is desirable that code generation enforces desired behaviors without altering the COTS interface. This calls for a specific design method. Guillet, De Lamotte, Le Griguer, Rutten, and Diguert (2014) apply UML modeling to system-on-chip reconfigurable design; DCS is used here as a tool for handling dynamic configuration switching on high performance dynamically reconfigurable FPGA systems.

This work advocates the use of the Discrete Controller Synthesis (DCS) technique (Marchand, Bournai, Le Borgne, & Guernic, 2000) in order to generate correct-by-construction code. A design method is proposed which highlights the synergy between usual design techniques, as advocated in Delaval et al. (2010). When COTS components are handled, it is desirable that code generation enforces desired behaviors without altering the COTS interface. This calls for a specific design method. Guillet, De Lamotte, Le Griguer, Rutten, and Diguert (2014) apply UML modeling to system-on-chip reconfigurable design; DCS is used here as a tool for handling dynamic configuration switching on high performance dynamically reconfigurable FPGA systems.

The second issue handled here concerns the possibly “spurious” behaviors added by automatically generated code. It is shown that despite its automatic generation, possible side-effects need to be subsequently assessed, with a possible impact on the validity of the result.

Finally, the industrial study is extended in order to illustrate the synergy between compositional reasoning and code generation in COTS-based design.

The rest of the paper is organized as follows: Section 2 recalls the backgrounds of the models and techniques used throughout the method proposed in this paper. The COTS-based design method is presented in Section 3. Section 4 illustrates an industrial application of the design method. An underlying tool-chain of this work is implemented in Section 5. The paper concludes with a summary and an outlook on the perspectives of this work in Section 6.

2. Background and notation

2.1. The Boolean finite state machine (BFSM)

This model is very useful in the context of hardware electronic embedded systems, because it is structurally and dynamically close to the handled hardware control-command systems. In this context, both Mealy and Moore FSM models are used by design engineers. For the sake of simplicity, this document only focuses on the Moore variant, without loss of generality.

BFSMs with outputs can represent communicating electronic components. The Boolean FSM model is defined as a tuple 

\[ M = (q_0, X, Q, \delta, out, \lambda) \]

where \( q_0 \) designates the initial state, \( X \) is a set of Boolean inputs, \( Q \) is the set of states of \( M \), \( \delta : \times X \times Q \rightarrow Q \) is the transition function, \( out \) is a vector of atomic Boolean propositions, and \( \lambda : Q \rightarrow \times out \) is a labeling function modeling the vector of outputs of \( M \).

For \( i = 1 \) to \( k : \lambda_i(q) \) iff \( out_i \) is true in state \( q \).

Example: Fig. 1 shows a small BFSM example, featuring a single input and a single output. The input variable is evaluated at each transition, while the output variable is assigned in each state. The state/transition graph representation is more convenient, for readability reasons.

The corresponding formal BFSM model is the following:

\[ q_0 = A \]

\[ X = \{input\} \]

\[ Q = \{A, B, C\} \delta(x, q) = \]

\[ \begin{align*}
(0, A) & \rightarrow A \\
(1, A) & \rightarrow B \\
(0, B) & \rightarrow B \\
(1, B) & \rightarrow C \\
(0, C) & \rightarrow C \\
(1, C) & \rightarrow A \\
\end{align*} \]

\[ out = \{output\} \lambda(q) = \]

\[ \begin{align*}
A & \rightarrow 0 \\
B & \rightarrow 0 \\
C & \rightarrow 1 \square \\
\end{align*} \]

BFSMs are naturally used for sequential modeling of dynamic behaviors. They are also used as building blocks for expressing concurrency. They are combined together according to the synchronous approach (Milner, 1983). This approach is the most natural in this framework, as the designs considered in this work are run by a unique common clock, as usual in electronic design. According to this restriction, all concurrent behaviors are executed synchronously: all transitions are triggered together. Communications are considered
atomic and instantaneous: they are triggered at each clock tick and are assumed to always terminate before the next clock tick.

2.2. The communicating synchronous product

This defines both a composite transition function, and an input–output interconnection for two BFSMs.

Let $\text{IOmap} : X \rightarrow X \cup \text{out}$ be a user-defined input/output mapping function. For $x = (x_1, ..., x_j), \forall x_i \in X$, $\text{IOmap}(x)$ denotes the vector obtained by application of $\text{IOmap}$ to each element of $x$.

The communicating synchronous product $\parallel_{\text{IOmap}}(M_1, M_2)$ between two BFSMs $M_1$ and $M_2$, which communicate according to a mapping $\text{IOmap} : X_1 \times X_2 \rightarrow X_1 \times X_2 \cup \text{out}_1 \cup \text{out}_2$ builds a new BFSM $M = (q_0, X, Q, \delta, \lambda, \text{out})$ according to the following rules:

- $q_0 = (q_0^1, q_0^2)$
- $X = \{x \in X_1 \cup X_2 | \text{IOmap}(x) = x\}$
- $Q = Q_1 \times Q_2$
- $\delta : \prod |X_1| \times |X_2| \times Q \rightarrow Q$ is defined as $\delta(\delta(\text{IOmap}(x_1)^1, x_2^1, q_1^1), \delta(\text{IOmap}(x_2^2), q_2^2))$
- $\text{out} = \text{out}_1 \cup \text{out}_2$
- $\lambda : Q \rightarrow \prod |\text{out}|$ is defined as $\lambda = (\lambda_1^1, \lambda_2^2)$

The $\parallel_{\text{IOmap}}$ operator is fundamental for both structural design, and for implementing concurrency, which is inherent to hardware embedded systems. It amounts to a formal/actual parameter mechanism which expresses interconnection and the communication between concurrent components, each modeled by a BFSM.

2.3. Hardware implementation of BFSMs

The formal BFSM model is a key representation: it is used by designers, as both a specification/implementation model, and is either used and/or produced by many design tools.

Design engineers represent BFSMs as procedures, using dedicated programming languages such as VHDL, which is one standard hardware description language. The problem is that such programming languages are general purpose. Many of them are not mapped to a formal semantics. For instance, the semantics of VHDL is explained textually, by a language reference manual (LRM). Besides, language constructs such as explicit timing, random aspect loops and recursiveness cannot be handled automatically in order to construct a BFSM. Thus, a language subset has emerged and become a standard (IEEE, 2004). By restricting the design code to this subset, BFSMs can be systematically and automatically extracted and fed to either simulation, hardware synthesis or formal verification tools.

2.4. Example. BFSM vs. VHDL

This example shows how state-based behaviors are modeled procedurally. A component has an interface, defined by an entity and a behavior defined by an architecture:

```vhdl
library ieee;
use IEEE.std_logic_1164.all;
entity sample is
port (clk : in std_logic;
       reset : in std_logic;
       input : in std_logic;
       output : out std_logic);
end sample;

The initial state is specified using a reset mechanism. The input and output variables are inferred from the entity port declaration. States and state transitions are inferred by analyzing the architecture; each assignment performed under a rising clock edge condition produces one state variable and one transition function. Output signals are assigned in each state:
```

![Fig. 1. FSM model extracted from VHDL](image-url)
The BFSM model extracted from the VHDL code given above is shown in Fig. 1, together with the formal BFSM model extracted.

This model extraction is handled efficiently by both industrial (Mentor-Graphics, 2010; Synopsys-Pro, 2014) and academic tools such as Alliance (Grenier & Pecheux, 1993).

2.5. Formal requirement specifications

Formal specifications are expressed either logically, as temporal logic formulae, written in the PSL (IEEE, 2005) standard language which is an extension of the temporal logic LTL (Pnueli, 1979) and CTL* (Hafer & Thomas, 1987), or operationally, as a “program” modeled formally by a BFSM and referred to as an observer.

In this work the “simple subset” of PSL (Foster, Marshcner, & Wolfsthal, 2005) is used to express functional requirements which are intended to be formally verified and/or simulated. The simple subset of PSL conforms to the notion of monotonic advancement of time. It contains besides the Boolean operators, a set of temporal operators: never, eventually, until, before, and also provides mechanisms to express desired behaviors as regular expressions. The restriction to the “simple subset” of PSL ensures (Foster et al., 2005) that properties within this subset can be assessed both by simulation and by formal verification. The mechanism which makes this possible is the automatic generation of equivalent observers, out of simple subset PSL formulae.

The observers used in this work express only safety assertions which can be constructed either manually or automatically (Oddos, Morin-Allory, & Borrione, 2009) from the simple subset of PSL specifications. Industrial frameworks like Mentor-Graphcis (2010) provide automatic translation of “simple-subset” PSL assertions into VHDL-coded observers. They are automatically extracted from meta-comment section provided in source code, as illustrated in the VHDL example above. The results presented in the sequel rely on this feature.

Table 1 illustrates the semantics of some PSL operators used throughout this paper, using the LTL syntax.

<table>
<thead>
<tr>
<th>PSL syntax</th>
<th>LTL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>next p</td>
<td>Xp</td>
</tr>
<tr>
<td>always p</td>
<td>Gp</td>
</tr>
<tr>
<td>never p</td>
<td>G¬p</td>
</tr>
<tr>
<td>eventually p</td>
<td>Fp</td>
</tr>
<tr>
<td>p until q</td>
<td>p U q</td>
</tr>
<tr>
<td>p until q</td>
<td>(p U q) or Gp</td>
</tr>
<tr>
<td>p before q</td>
<td>(¬q U (p ∧ ¬q)) ∨ G(¬q)</td>
</tr>
</tbody>
</table>

For example, the PSL requirement
alwaysvalid → (request before acknowledge)

states that each time the variable valid is true, the variable request should become true before the variable acknowledge becomes true but not necessarily in the exact previous instant. If request never becomes true, neither should acknowledge.

2.6. Observer-based requirements

The formal framework and tool provided by Oddos et al. (2009) has been used in this work in order to translate (simple subset) PSL assertions into observers. The reader is directed to this reference for further information about this topic. The principles of this technique are recalled below.

Formal requirements are monitored by observing sequences of values. Fig. 2 illustrates an observer, modeled by a BFSM. Its input variables are intended to be mapped to the inputs, the outputs and possibly the states of the design at hand, and its outputs are always the same:

- pending is true if a sequence of values is being observed and no decision can be made so far about the requirement violation;
- error is true if a sequence of values observed has violated the requirement.

Any occurrence of ¬pending ∧ error means that a requirement violation has been observed. Hence, all the requirements translatable into observers are assessed by evaluation of a generic assertion:

p : never¬pending ∧ error

on the synchronous product between the design model and the observer model, as shown in Fig. 2.

An important advantage of this observer-based design pattern is its genericity with respect to design tools: it can be simulated, formally verified, or even implemented in hardware when needed. Besides, observers can also be built manually, which gives more flexibility for modeling requirements.

2.7. Control-command COTS (C-C COTS)

These are the basic building blocks considered in this work. In the sequel the “COTS” abbreviation refers to the control-command COTS. A COTS interacts with its environment, via its inputs and outputs. The environment of a COTS can be either physical, namely sensors and actuators, or logical, a collection of COTS, or both. Fig. 3 shows that COTS can be interconnected together, in order to achieve complex control-command functions, and that they can also be connected to an operative (physical) part, issuing adequate commands and reading its state through sensors. A user interface, symbolized by a control panel, can also be a part of a design flow. Human operators issue requests, by acting on specific controls, and receive acknowledgments to their requests.

In the end, by assembling a collection of COTS, designers build control-command systems, intended to interact with a physical environment exclusively, made of sensors and actuators: the term “control” indicates reading the current state of the physical environment, whereas the term “command” indicates commanding the physical environment.

A stand-alone COTS C is defined as a 4-tuple C = (I, F, A, G), where I is the COTS’ input–output interface, expressed as a collection of Boolean variables, F is the behavioral model of the COTS expressed as a BFSM, A is a set of assumptions on the expected behavior of the environment of C, and G is a set of guarantees on the behavior M of C. Both assumptions and guarantees are expressed on interface variables. The four elements of a stand-alone COTS tuple are shown in Fig. 4.

![Fig. 2. System under observation.](image-url)
Both the assumptions and the guarantees are expressed formally, either as PSL formulae or as hand written observers, modeled as BFSMs. It is said that the behavioral model $M^C$ of a COTS $C$ satisfies a guarantee $g \in G^C$ provided that an assumption $a \in A^C$ holds. This is denoted:

$$M^C, a \models g$$

As both assumptions and guarantees are expressed as an extension of temporal logic (PSL), their semantics is identical to that of temporal logic in general and of PSL in particular; both $a$ and $g$ evaluate to their truth value at the initial state of $M^C$.

The COTS's behavior is expressed as code, using a standard and/or proprietary framework. All the components handled in this work are automatically translatable into BFSMs. Hence, control-command COTS-based designs are produced by assembling COTS together, taking into consideration their interface, and their assume/guarantee sets. This aspect is developed in Section 3.

Note that a COTS is considered rather a “perfect” component than a “perfect” one; it probably has hidden bugs, and building designs out of existing COTS also amounts to potentially mixing unwanted behaviors from each building block.

The assume/guarantee components of each COTS are key features in COTS-based design; the quality of these sets is crucial for obtaining the time and cost benefits, that component-based design techniques claim to offer. A “good quality” assume or guarantee set can be defined as a set which contains the necessary and sufficient assertions describing either the COTS environment or its behavioral model. The quality of a COTS also involves the way guarantees are provided: either formally, or by simulation; thus, most of the time, guarantees are rather confident assertions, than certitudes. This quality target remains difficult to reach in practice, because it requires an important and manual design effort which cannot always be afforded.

### 2.8. The model checking of formal requirements

This technique has become a key one within industrial design flows as it affirms its success in detecting design “corner-case” errors, which are difficult to uncover by simulation (Clarke, 2008; Fix, 2008). It achieves an exhaustive exploration of the design’s state space, in order to check the satisfaction of a requirement written in temporal logic (Clarke & Emerson, 1982). The first significant breakthrough of this technique, known as Symbolic model checking, developed in McMillan (1992) has been proposed during the early 90s. It relies on Binary Decision Diagrams (Bryant, 1986) for achieving an efficient representation of large sets of states. In order to tackle more and more complex designs, this technique has been intensively investigated and improved by the research community. The latest model checking techniques proposed (Eén, 2005; Hassan, Bradley, & Somenzi, 2012) rely on SAT engines (Eén & Sörensson, 2003), and the most efficient commercially available tools are even able to switch between BDDs and SAT in order to enhance their performance. The success of this technique is demonstrated by the fact that for about twenty years, the model checking application principles have remained quite the same; almost all research and development efforts follow the same direction: its performance enhancement.

In the context of this work, the model checking technique is advocated in order to build the assume/guarantee sets of a given COTS, and also to assess new assertions for COTS assemblies.

### 2.9. The discrete controller synthesis (DCS)

This technique enforces the satisfaction of a safety assertion $P$ on a given BFSM model $M$ by attempting to make invariant the greatest subset of states of $M$ which satisfy $P$. The input set of $M$ is divided into two disjoint subsets: controllable $X_c$ and uncontrollable $X_u$ inputs. The target set satisfying $P$ is made invariant by disabling all the transitions of $M$ leading out of it. This is achieved by generating a supervisor that assigns adequate values to the controllable inputs $X_c$. The supervisor is defined as:

$$SUP = \{((X_c, X_u), q) | \delta((X_c, X_u), q) \in IUC\}$$

The DCS proceeds in two steps: (1) computation of the invariant under control (IUC) set and (2) computation of the supervisor.

The computation of IUC is performed recursively by successively finding the set of controllable predecessors of a given set of states $E \subseteq Q$. This step is implemented by the CPRED operator:

$$CPRED(E, \delta) = \{q \in Q | \forall x_{uc} \in E \implies |X_{uc}|, \exists x_c \in E \implies |X_c|, \exists q' \in Q : q' = \delta((x_c, x_{uc}), q) \land q' \in E\}$$

In other words, the state $q$ is a controllable predecessor of a state $q' \in E$ if for any uncontrollable value $x_{uc}$, there exists a controllable value $x_c$ such that the transition function $\delta$ leads to $q'$.

The resulting invariant under control set $IUC$ is the fixed point of the equation:

$$IUC_0 = \{q | P \text{ is true in } q\}$$

$$IUC_{i+1} = IUC_i \cap CPRED(IUC_i, \delta)$$

A supervisor does not exist if the $IUC$ set is empty or if it does not contain $q_0$.

#### 2.9.1. DCS for Hardware design

The supervisor provided by DCS is implemented as a characteristic function (Marchand et al., 2000):

$$SUP : \Box |X_c| + |X_{uc}| \mapsto Q \rightarrow \Box$$
defined as
\[ S\overline{P}(x_c, x_{uc}), q) = 1 \text{ iff} ((x_c, x_{uc}), q) \in S\overline{P} \]

Hence, the actual control of \( M \) requires solving the equation
\[ S\overline{P}(x_c, x_{uc}), q) = 1 \quad (2) \]
continuously, for each transition of \( M \), considering \( x_c \) as unknown variables. However, a hardware (FPGA) implementation of this control loop requires that the value of each controllable variable \( x_{ci}, i \in \{1, \ldots, |X_c|\} \) be computed by an appropriate expression, for each transition of \( M \). This amounts to a symbolic resolution of Eq. (2). The structure of \( S\overline{P} \) is totally unfit for the straightforward resolution of this equation.

The supervisor decomposition technique presented in Dumitrescu, Ren, Piétrac, and Niel (2008) is used in order to solve Eq. (2) symbolically. This approach yields the control architecture presented in Fig. 5. This technique amounts to a parametric decomposition of a characteristic function, defined on the Boolean domain. It is carried out following two operations, applied in sequence, for each controllable variable.

Step 1: Split the expression \( S\overline{P} \) in order to separate the unknown variable \( x_{ci} \) from the sub-expressions of \( S\overline{P} \) free of \( x_{ci} \). This is performed by rewriting Eq. (2) according to Boole’s expansion theorem with respect to \( x_{ci} \). Let \( S\overline{P} | x_{ci} = e \) be the expression obtained by substituting the Boolean expression \( e \) for the variable \( x_{ci} \) in \( S\overline{P} \). These expressions are also known as co-factors of \( S\overline{P} \) with respect to \( x_{ci} \). The following holds according to Boole’s expansion theorem:
\[ S\overline{P} = \neg x_{ci} \land S\overline{P} \big| x_{ci} = 0 + x_{ci} \land S\overline{P} \big| x_{ci} = 1 \]

Step 2: Solve the control equation (2) symbolically for variable \( x_{ci} \). For any value of \( i \), the values of \( x_{ci} \) which satisfy this equation are detailed in Table 2.

| \( S\overline{P} \big| x_{ci} = 1 \) | \( S\overline{P} \big| x_{ci} = 0 \) | \( x_{ci} \) |
|-----------------|-----------------|---------|
| 0               | 1               | 0       |
| 1               | 0               | 1       |
| 1               | 1               | 0 or 1  |

\[ \exists x_{ci} \text{ when } S\overline{P} \text{ is satisfiable.} \]

Step 3: Substitute \( f_i \) for \( x_{ci} \) in \( S\overline{P} \). The resulting expression \( S\overline{P} \big| x_{ci} = f_i \) is a starting point for a subsequent application of the Split/Solve/Substitute steps defined above.

Let \( \tau: \{1, \ldots, |X_c|\} \to \{1, \ldots, |X_c|\} \) be a permutation defining a user-defined solving order for the controllable variables \( x_c \). The complete resolution of this equation system is performed by computing the following symbolic expressions. For all \( i = \tau(1), \tau(2), \ldots, \tau(|X_c|) \):
\[ f_i = \neg S\overline{P} \big| x_{ci} = 0 \land S\overline{P} \big| x_{ci} = 1 \]
\[ \lor x_{ci}^{env} \land S\overline{P} \big| x_{ci} = 0 \land S\overline{P} \big| x_{ci} = 1 \]

where \( S\overline{P}_1 = S\overline{P} \text{ and } S\overline{P}_1 = S\overline{P} \big| x_{ci} = f_i \).

The resulting controller is a vector \( \hat{C} \) of \( |X_c| \) Boolean functions:
\[ \hat{C} = \langle g, x_{uc} \cup x_{ci}^{env} \cup q, \hat{X}_c, x_{uc}, \hat{C} \rangle \]

The model of the controlled system is built by mapping the outputs of \( M^\hat{C} \) to the controllable variables \( X_c \): \( M = \| \text{map} \| (M, M^\hat{C}) \) where \( \text{map} : X_{uc} \cup X_{ci} \cup Q \to X_{uc} \cup X_{ci} \cup Q \) is defined as an identity function in this particular situation, merely translating variable names correspondence.

It is important to note that the choice of \( \tau \) may have an impact on the resulting controller. It establishes an evaluation order, in order to handle the assignment of controllable variables in case of control non-determinism, by taking into account the choices made for the others. Thus, variable \( x_{ci}(X_{ci}) \) is always evaluated first, because its controller expression is computed last, and does not depend on the other controllable variables.

All choices of \( \tau \) are valid, but not always equivalent. Finding a convenient order amounts to a choice, made by the designer, prior to the supervisor decomposition. The only guideline available to make this choice is expressing an evaluation priority between controllable variables.

As shown in Fig. 5, the control architecture obtained interferes with the environment by filtering the values of the controllable inputs. This situation is globally undesirable but acceptable, cautiously; however, this issue induces additional design constraints, developed in Section 3.4.

### 2.9.2. A DCS illustrative example

Consider the state-based design shown in Fig. 6. Let a property \( P = \text{always} (\neg \text{ck} \lor \text{ack}) \) be the target requirement to enforce using DCS, by controlling the input variable \( go \). The IUC computation algorithm gives the
following results:

\[ \begin{align*}
IUC_0 & = \{A, B, C\} \\
IUC_1 & = \{A, C\} \\
IUC_2 & = \{A, C\}
\end{align*} \]

The final IUC set is \( \{A, C\} \). The generated controller \( \tilde{c} \) assigns the controllable variable \( \text{go} \), so that the controlled system always remains inside the set of states IUC, as illustrated in Fig. 7.

### 2.10. The Environment-aware DCS (EDCS)

The DCS algorithm presented above does not support the specification of environment assumptions. In order to handle this additional information, a variant of the DCS algorithm is proposed, called Environment aware DCS. It redefines the computation of the controllable predecessors by assuming, that at each step, the uncontrollable inputs satisfy the environment assumptions. Each assumption is modeled as a safety assertion \( \alpha \in A^\tau \) concerning the uncontrollable inputs. It is translated into an invariant \( A : Q \times \mathbb{I}^{\left| X_{uc}\right|} \rightarrow \mathbb{B} \) characterizing the set of all the transitions of \( M \) satisfying \( \alpha \):

\[ \begin{align*}
A(q, x_{uc}) & = \exists x_c \in \mathbb{I}^{|X_c|} : \alpha \text{ is true in state } \delta(q, x_c, x_{uc})
\end{align*} \]

During EDCS computation, \( A \) is supposed to be always true. The computation of the environment-aware controllable predecessors is defined as follows:

\[ \text{CPRED}_{env}(\mathcal{E}, \delta, A) = \{ q \in Q | \forall x_{uc} \in \mathbb{I}^{|X_{uc}|}, \exists x_c \in \mathbb{I}^{|X_c|}, \exists q' \in Q : (q' = \delta((x_{uc}, x_c), q) \land A(q, x_{uc})) \rightarrow q' \in \mathcal{E} \} \]

The recursive application of \( \text{CPRED}_{env} \) produces an invariant under control set under an environment assumption. This variant of the DCS technique is less "pessimistic" with respect to the uncontrollable input variables, and thus less restrictive as it assumes a correct environment behavior.

#### 2.10.1. EDCS illustrative example

Consider the example presented in Fig. 6. The same requirement \( P \) is synthesized using EDCS, under a sample environment assumption \( a \), stating that the uncontrollable \( \text{req} \) must be asserted when the system is in state \( B \):

\[ a = \text{always}(B \rightarrow \text{req}) \]

This assumption is modeled by an observer FSM illustrated in Fig. 8 where the vector of outputs is \((\text{pending}, \text{error})\). The EDCS application computes the invariant under control \( IUC = \{A, B, C\} \) and generates a controller which keeps the system inside the set \( IUC \) as shown in Fig. 9. Unlike conventional DCS, state \( B \) is not pruned, as \( \text{req} \) is supposed to be asserted whenever this state is active, and thus, due to this assumption, the error state \( E_1 \) is not reached. Thus, EDCS generates a controller to prevent the system from reaching the state \( E_2 \).

### 3. The safe COTS-based design method

The method proposed in the sequel relies on the conjunction between traditional design techniques, namely simulation, formal verification, and the DCS. These techniques offer new design possibilities yet to be exploited in practice in the context of the work. It develops a synergy between these techniques. An overview is presented in Fig. 10. Designers carry out in sequence the following steps: modeling (Step 1), either by writing new code or by reusing existing COTS; verification comes next (Step 2), for assessing the correctness of either new code, or COTS assemblies; the specificity of the design method proposed lies in the automatic correction of some design errors, using DCS (Step 3). Formal verification and simulation are used subsequently (Steps 4 and 5), in order to validate the generated controlled implementation. These steps are detailed below.

#### 3.1. Step 1: Modeling

Step (1) Modeling of Fig. 10 refers to the manual and intellectual work performed by design engineers in order to translate functional requirements into a new control-command function. It relies on the systematic availability of a library of reusable components, referred to as a COTS library. The modeling work amounts to either

- reusing existing COTS in the library, according to their sets of assure/guarantee properties, or
- writing new code

or both.

The efficiency of the modeling work is generally measured by the time it requires. It is commonly accepted that code reuse is extremely desirable in order to reduce production delays. Ideally, new control-command functions should be built on top of existing ones, either purchased or available from the COTS library.

#### 3.1.1. The COTS assembly

Is the act of composing COTS components together, in order to produce a new behavior. This act amounts to finding a matching
Fig. 9. Controller synthesis using EDCS.

Indeed, the interface of the compound result cannot grow bigger than the interfaces of all components. Besides, due to possible matchings between assumes and guarantees of components C1 and C2, some assumptions may be dropped. The same may occur for the resulting set of guarantees.

Formally, $A_{\text{asm}}$ should be the (smallest) set of necessary assumptions; the same applies for the set $G_{\text{asm}}$. Yet, to the best of author’s knowledge, no support exists for such a systematic construction, which is why all matchings between assumptions and guarantees are performed manually, by (human) designers. The reliability of this process is highly dependent on the designer’s skill and is totally random.

3.1.4. Cyclic reasoning

This situation is highlighted in Fig. 11, patterns (5,6,8,9). It is characterized by a cyclic dependency between assumes and guarantees which, on a two-component assembly pattern, is expressed as follows:

$$M_{\text{asm}} = M_{\text{C1}} \cup \text{IoMap} M_{\text{C2}}$$

When two or more COTS are assembled it may result in assume/guarantee incompatibilities which may appear at the moment the COTS are assembled. Consider the assembly patterns highlighted in Fig. 11. The following issues need to be considered.

3.1.2. The incompatibility between environment assumptions

This may occur in the assembly pattern (4,7), highlighted in Fig. 11. According to this pattern, components C1 and C2 share a common environment behavior through the input sets $x_{1,2}$ and $x_{2,1}$. Each COTS comes with its environment assumptions $A^{C1}$ and $A^{C2}$. In this situation, it must be ensured that

$$\exists (a_1, a_2) \in A^{C1} \times A^{C2} \text{ such that } \neg (a_1 \land a_2)$$

Usually this check is performed manually. If a pair of contradictory assumptions exists, then it is considered that the COTS assembly in question is not valid. Otherwise, it can be concluded that in this particular situation

$$A_{\text{asm}} = A^{C1} \cup A^{C2}$$

$$G_{\text{asm}} = G^{C1} \cup G^{C2}$$

Hence, the local guarantee sets are conserved on the assembly.

3.1.3. The contradiction between guarantees and environment assumptions

This may occur when a component’s outputs drive the inputs of another component. This is highlighted in Fig. 11, patterns (2,3,7). In these situations, it must be ensured that there is no guarantee in component C1 which contradicts an assumption in C2:

$$A_{\text{asm}} = (A^{C1} \cup A^{C2}) \setminus \{a_1 \in A^{C1} \mid \exists g_2 \in G^{C2} : g_2 \rightarrow a_1\}$$

$$G_{\text{asm}} = G^{C1} \cup G^{C2}$$

This check is achieved manually, on the sets of assumptions and guarantees. If a guarantee is shown to be broken, the COTS assembly is not valid.

However, when the above check is successful, it can be concluded that the COTS assembly preserves all local guarantees. This situation is the most desired in COTS-based design: after the COTS assembly operation the guarantees do not need to be re-assessed. Besides, all the environment assumptions required by C1 and implied by the guarantees of C2 do not need to be assumed anymore. In this situation the assembly operation yields the following assume/guarantee sets:

$$A_{\text{asm}} = A^{C1} \cup A^{C2} \setminus \{a_1 \in A^{C1} \mid \exists g_2 \in G^{C2} : g_2 \rightarrow a_1\}$$

$$G_{\text{asm}} = G^{C1} \cup G^{C2}$$

The behavioral model of the COTS assembly is the synchronous product (Cassandras & Lafortune, 2010) of the BFSM of each individual COTS:

$$M_{\text{asm}} = M^{C1} \circ \text{IoMap} M^{C2}$$

between the guarantees of one component and the assumptions of the other. The interaction which is established has a direct impact on the set of guarantees: they cannot be conserved and need to be re-assessed. This is why the fact of assembling COTS produces a new component which cannot yet be considered as a COTS. Existing assumptions of one component may be implied by the guarantees of the other and thus need not be assumed anymore. But they may also be contradicted, in which case they cannot be assumed anymore. The COTS assembly operation is denoted by the operator $\bowtie$. It is performed according to the user-defined structural mapping $\text{IoMap}$ which can yield different assembly patterns, as shown in Fig. 11.

Given two COTS $C_1$ and $C_2$, the COTS assembly $C_1 \bowtie C_2$ has the following properties:

$$F_{\text{asm}} \subseteq F^1 \cup F^2$$

$$A_{\text{asm}} \subseteq A^{C1} \cup A^{C2}$$

$$G_{\text{asm}} \subseteq G^{C1} \cup G^{C2}$$
In other words, $C_1$ provides guarantees $g_1$ that enable assumptions $a_2$ of $C_2$ and vice versa. This circular dependency cannot be exploited in a COTS assembly operation, because it can generally not be concluded that $M^{C_1} \parallel M^{C_2} \models g_1 \land g_2$

This situation requires the re-assessment of $g_1$ and $g_2$ for the composition $M^{C_1} \parallel M^{C_2}$.

Such circular reasoning situations are to be handled separately through formal verification after the assembly. Circular reasoning can only be applied in some particular situations (Henzinger, Qadeer, & Rajamani, 1998; McMillan, 1999). In such situations, the actual proof requires an important amount of user assistance. The technicality of such proofs is both error prone and beyond the skills of average design engineers.

According to these compositional reasoning patterns, a safe COTS assembly can be defined as follows.

**Definition 1 (Safe COTS assembly).** A COTS assembly is safe if and only if the following hold:

- there is no incompatibility between environment assumptions;
- there is no contradiction between guarantees and environment assumptions;
- it does not contain cyclic assumption/guarantee dependencies: $\exists a_1, a_2 \in A^{asm}, \exists g_1, g_2 \in G^{asm}: a_1 \rightarrow g_1 \rightarrow a_2 \rightarrow g_2 \rightarrow a_1$

A COTS assembly operation creates a new component, according to new requirements, expressed as new assertions, possibly relying on new assumptions. New assertions need to be evaluated, and according to this evaluation they can become guarantees. The evaluation process is performed during the step 2 of the design method.

### 3.2. Step 2: automatic error detection

The COTS assembly is formally verified against a set of requirements $P^{asm}$ expressed manually by designers. The set $P^{asm}$ is defined as

$$P^{asm} = P_s \cup P_l \cup G^{asm}_{gb}$$

where $P_s$ is a set of safety properties, $P_l$ is a set of liveness properties, and $G^{asm}_{gb}$ is a set of possibly broken guarantees

$$G^{asm}_{gb} = (G^{s}_1 \cup G^{s}_2)$$

in case the COTS assembly is not safe. All these requirements are expressed as temporal assertions formalized in PSL.

If all elements of $P^{asm}$ are true, the design process can reiterate from step 1. Should the opposite occur, two situations need to be considered:

- a safety assertion is false; it can be attempted to correct it automatically as highlighted in step 3 below;
- a liveness assertion is false; it cannot be automatically corrected, and requires (manual) component redesign.
3.3. Step 3: automatic error correction

The automatic error correction is carried out by using EDCS, in order to enforce the safety assertions $P_i$ identified to be violated by the COTS assembly.

Finding controllable inputs is a necessary step in order to apply EDCS. This requires a choice among the available input variables. However, this choice cannot be done at random: the nature of each input variable should be considered.

Typically, the designer is able to distinguish between the following situations:

- input variables driven by physical sensors with hard reactive processing constraints. The processing of such information should follow the rhythm of the environment. EDCS should consider such inputs as uncontrollable;
- input variables driven by physical sensors with interactive processing constraints. When processing such inputs, it can be assumed that the physical environment can wait for a variable but reasonable amount of time. Typically, human users accept interactive processing constraints. Such inputs can be controlled by EDCS;
- inputs carrying data. The control of such inputs is delicate, because it amounts to possibly altering the content of a “message” sent between two components in a way that is unpredictable for designers. This is why in this work, these inputs are chosen to be uncontrollable.

By exploiting the counterexample provided by the model checking tool, the designer can construct a list of candidate controllable signals. The counterexample is a simulation trace showing the sequence of values of the input/state/output variables leading to the violation of a property. The input variables displayed by a counter-example are clearly involved in this violation. It can be attempted to prevent this situation, by adequately controlling these inputs. This is summarized by the following procedure:

Construction of the controllable signals $X_c$ set

1. $X_c^{final}$ = the complete set of signals provided by the model checking counterexample.
2. remove from $X_c$ the uncontrollable signals by following the rules:
3. Begin removing:
4. Every input variable intended to be driven by a sensor with hard reactive constraints must be eliminated from the $X_c$ set;
5. Every input variable representing data must be eliminated from the $X_c$ set;
6. Every input variable representing an alert or alarm information must be eliminated;
7. Every state variable must be eliminated;
8. Keep all the remaining signals and construct the list of controllable signals $X_c$.

3.4. Implementation of the control loop

Even though the EDCS-based approach for enforcing the satisfaction of safety requirements is attractive, its actual implementation is delicate due to one main reason: the designation of controllable inputs of a COTS or a COTS assembly is not intuitive for design engineers. The procedure above provides a way to cut down the number of controllable candidates; however the designer’s knowledge about the system remains preponderant in the choice of the controllable signals.

Three typical implementation patterns can be identified. They are structurally identical, but the behaviors they implement and the requirements they should satisfy are different, according to the knowledge available about their function: a general case, where no a priori knowledge exists to validate the controller, and two cases where the reactive constraints are either hard, requiring bounded time reactions, or soft, requiring finite time reactions.

Recall that the hardware system behaviors are modeled following the sample-driven paradigm, where a unique clock exists for the whole system. The input values are sampled at each clock tick. In the examples presented below, a clock tick is associated to a rising edge of the clock signal.

3.4.1. The general case

This control architecture implements the classical closed-loop control. The input variable $x$, designated as a controllable input, is driven by the controller, according to the property enforced, and the value desired by the environment. There is no a priori knowledge about the expected behavior of $x^{env}$. Hence, the possible behaviors of the controlled design are exemplified in Fig. 12; the controller can either forward the input value $x = x^{env}$ or mask it $x = -x^{env}$. It is up to the designer to judge if it is acceptable to filter such inputs.

3.4.2. Controllable inputs with hard reactive constraints

Such inputs are driven by physical sensors. Let $x^s$ be such an input. In practice, two possibilities exist:

- make an EDCS attempt by considering $x^s$ as uncontrollable. If a control solution is found, continue with the verification process in step 4;
- if no control solution is found, then attempt an EDCS step by considering $x^s$ as controllable. The only tolerated action for the controller is delaying the active values of $x^{env}$ for an acceptable amount of time, short enough for sampling the sensor activation. This is shown in Fig. 13. A functional assessment is required, in order to establish that each time $x^{env}$ rises, it shall eventually be sampled by the controlled system. However, in practice the input $x^{env}$ should only be delayed for a bounded amount of time. Thus, a performance estimation on the physical FPGA implementation of the controlled design is also required, in order to determine that bound, and assess if the actual system speed is sufficient.

Due to its delicate implementation, this implementation pattern is not recommended. However, if it cannot be helped otherwise (no control solution is found), a compromise solution can consist of either sampling the freshest sensor value, or buffering the sensor information so that every input value is memorized until it is processed. Of course, even buffering requires dimensioning, but this can be achieved statically, using the FIFO (First In First Out) theory.

3.4.3. Controllable inputs with soft reactive constraints

“Soft” reactivity is most often sufficient for achieving communication between components; such interactions are required to be reliable and to last a finite time. Reliability is guaranteed by systematic master–slave synchronization mechanisms. The 4-phase handshake protocol is both a generic and representative communication mechanism in hardware design. It is implemented by a pair of Boolean signals: a request and an acknowledge. The handshake protocol starts when the request is activated. The acknowledge is then activated, followed by the request de-activation, and finally by the acknowledge deactivation. This sequence is called a transaction.
3.5. Step 4: the formal verification of the controlled system

The formal verification in this step has two main motivations. As the generated controller interferes with the environment of the controlled system, it can be in contradiction with the environment assumptions of this system. The first objective of this verification step is to ensure that the liveness environment assumptions are not broken, and thus, that the liveness guarantees are preserved. Second, it must be ensured that the controller does not introduce spurious behaviors. This problem is presented below.

3.5.1. The "event invention" phenomenon

A controller which acts upon the request input is not aware of the notions of activation or deactivation, but only manipulates the values 1 or 0 of this input. At some moments, the value 1 can be forbidden (and thus the value 0 is forced), or vice versa. However, these two situations are not symmetric: in the first case, the transaction does not start, while in the second, the transaction is forced, or "invented"! This is illustrated in Fig. 15. Usually, transactions also carry data, and hence such a situation does not make sense. Obviously, this is unacceptable. This requires to make sure a posteriori that the controller never "invents" transactions, and if it does, invalidate the control solution.

Hence it is vital to formally ensure the absence of "event-invention" phenomena. The expression of this requirement for a controllable variable \( x \) needs to mention systematically \( x^{\text{env}} \), which is generated by (E)DCS, as explained in Section 3. However, the variable \( x^{\text{env}} \) does not exist at the moment DCS starts. It is not possible to mention its name to express requirements over the resulting controller. This is why the event invention phenomenon cannot be forbidden, but only detected by model checking.

In this situation, the decision procedure designates as controllable the request input, which is a part of the synchronization mechanism.

For this specific case, the desired control should implement one among the following behaviors: either prevent a transaction from starting if its beginning is likely to break the requirement to enforce, or let it start otherwise. The transaction start is triggered by the environment, through the request\(^{\text{env}} \) input variable. This value should be either forwarded or delayed by the controller. Fig. 14a shows the structural separation between the request\(^{\text{env}} \) and request signals. Incoming active request\(^{\text{env}} \) values are filtered by the controller according to the control objective, as exemplified in Fig. 14b.

3.5.2. Detection of "event inventions"

This behavior is simply checked by the PSL property:

\[
\text{NO\_EVENT\_INVENTION} : \text{always}(\text{request}^{\text{controlled}} \land \neg \text{request}^{\text{env}})
\]

In complement, it must also be established that the controller is not too restrictive, by delaying transactions forever. Thus, once a transaction starts, it must be acknowledged within finite time:

\[
\text{FINITE\_TRANSACTION} = \text{always}(\text{request}^{\text{env}} \rightarrow \text{eventually acknowledge})
\]

where stable is a built-in PSL operator: stable\((x)\) evaluates to true in every cycle where variable \( x \) did not change its value with respect to the previous cycle.

If this verification step is successful, the resulting controlled COTS can be considered as valid.

3.6. Step 5: simulation

The last step in the design method is the simulation of the controlled system. It ensures that the controlled system is still operating as needed. Since in the error correction step some behaviors were removed by the EDCS, the designer needs to acquire supplementary insight on the controller decisions; in particular, as a complement to formal liveness verification, simulation can witness the fact that the controller is not too restrictive, which means it does not suppress required system’s behavior. In this step the designer can observe the evolution of the resulting controlled system before the final hardware implementation. Beyond the subjective need of visualizing the system behavior before implementing it, simulation is used when the formal verification reaches its complexity limits. The same temporal properties can be checked by guided simulation, with much better scalability. Of course, the result is not exhaustively established, and
the validity of a “proof” depends on the bound chosen for the
simulation time, and thus for the simulation efforts.

4. Industrial application

The method proposed above is applied to a passenger access
control-command system, featuring three COTS: door, open
authorization, and filling-gap. They are illustrated in Fig. 16. Each
COTS is connected to a physical component in the train (physical
door and filling-gap, speed sensor) and to the driver control panel.

This case study was led by the framework of FUI (Fonds unique
interministériel) FerroCOTS (2009–2012) project (Jadot, 2009)
supported by i-Trans competitive pole and under the responsi-
bility of Bombardier Transport.

The passengers access system is designed by following the five
steps of the proposed method.

4.1. Modeling the COTS and the properties

The Door COTS Model is 4-tuple
\[ C^d = (F^d, M^d, A^d, G^d) \]
where
\[ F^d = \{ \text{req\_open, req\_close, open\_auth, sns\_open, sns\_close, sns\_obst, cmd\_open, cmd\_close, ack\_open, ack\_close} \} \]

The behavior \( M^d \) of the Door COTS is modeled by a BFSM shown in
Fig. 17. The train conductor issues open or close requests via the
\( \text{req\_open} \) or \( \text{req\_close} \) signals and the control-command answers
with a corresponding acknowledge, once the physical part has had
the expected reaction. The door opening is conditioned by an
opening authorization, given by the train conductor. The output
vector is \( \{ \text{cmd\_open, cmd\_close, ack\_open, ack\_close} \} \). It models
the commands issued to the doors and the acknowledge mechan-
isms for the open and close operations. The states of this COTS
reflect the states of the physical part. If an obstacle is detected a
dedicated sensor \( \text{sns\_obst} \) is activated, and the closing operation is
suspended for a few seconds. Once this delay has passed, a timer
(counter) sets the variable \( \text{time} \) to true. The closing operation resumes. Doors are considered closed as soon as the dedicated

sensor \( \text{sns\_close} \) is activated. An acknowledge is sent via \( \text{ack\_close} \).
The opening behavior is very similar. The environment assumption
required for the correct behavior of the Door COTS are given by the
set \( A^d = \{ a_1^d, a_2^d, a_3^d, a_4^d \} \), where \( a_1^d, a_2^d \) express sensor liveness and \( a_3^d, a_4^d \)
the absence of request cancellation. Assumption \( a_5^d \) states that the
doors opening authorization holds long enough to be exploitable:
\[ a_1^d = \text{always eventually}(\text{sns\_open}) \]
\[ a_2^d = \text{always eventually}(\text{sns\_closed}) \]
\[ a_3^d = \text{always req\_open} \rightarrow \text{stable(req\_open) until ack\_open} \]
\[ a_4^d = \text{always req\_close} \rightarrow \text{stable(req\_close) until ack\_close} \]
\[ a_5^d = \text{always rose(open\_auth)→next(stable(open\_auth) until ack\_close)} \]

The guarantees of the Door COTS are given by the set
\[ G^d = \{ g_1^d, g_2^d, g_3^d \} \]. Guarantees \( g_1^d, g_2^d \) express the fact that if the door
is requested to open or close the request is finally treated. Guarantee \( g_3^d \) states that the doors cannot be opened if the
conductor did not authorize it previously. They are modeled by

Fig. 17. Door COTS behavioral model.

Fig. 16. The COTS and their physical environment.
COTS are given by the set assumptions required for the correct behavior of the OpenAUTH doors can be opened at will, either by the conductor or by the opening is authorized. As long as the authorization holds, the authorization, by issuing a req as the train runs under 3 km/h, if the driver activates the opening forever: guarantees of the Filling-gap COTS are given by the set McC, where Gfg 1 and Gfg 2 express sensor liveness and Gfg 3 and Gfg 4 the absence of request cancellation: guarantees of the filling-gap COTS are given by the set Gf = (gF 1, gF 2, gF 3). They express the fact that if the filling-gap is requested to deploy or withdraw the request is finally treated. They are modeled by the PSL assertions: the relationships between A 4 and G 4 are the following: the relationships between A 4 and G 4 are the following: The Filling gap COTS is similar to the Door COTS: Its interface is Its behavioral model Mfg is a BFSM illustrated in Fig. 19, its output vector is (cmd_deploy, cmd_withdraw, ack_deploy, ack_withdraw). Its function consists in deploying and retracting the physical filling gap according to the requests sent by the conductor via the inputs req_deploy and req_withdraw. The environment assumption required for the correct behavior of the Filling-gap COTS are given by the set Afg = (aF 1, aF 2, aF 3, aF 4), where aF 1 expresses sensor liveness and aF 2, aF 3, aF 4 the absence of request cancellation: The guarantees of the filling-gap COTS are given by the set Gf = (gF 1, gF 2, gF 3). They express the fact that if the filling-gap is requested to deploy or withdraw the request is finally treated. They are modeled by the PSL assertions: The relationships between Afg and Gfg are the following: 4.2. Modeling the COTS assembly The OpenAUTH/Door/Filling-gap assembly C 0 || Cd || Cfg is modeled as follows: For the sake of readability, it has been chosen to make a total name correspondence between interface variables. This is why in this particular case, 10map is merely an identity function.
Fig. 20. Observer $M^{\text{pasm}}$ modeling $P^{\text{pasm}}$.

Fig. 19. Filling-gap COTS behavioral model.

4.4. Automatic error detection

The variable open_auth can become hidden through component assembly. No explicit requirement exists, so there seems no reason to leave it visible.

The assumption $a^3_f$ appears to be implied by guarantee $g^3_f$, which is why it needs not be assumed anymore. It ought to be emphasized that this is established manually, according to compositional reasoning. The only way to make sure $a^3_f$ can be dropped would be to formally verify that $M^p = a^3_f$.

Apart from COTS OpenAUTH and Door, which have been assessed above, there is no other interaction between the remaining COTS. It can be concluded that the set of resulting guarantees $G^{\text{pasm}}$ accumulates the local guarantees of each COTS.

4.3. Modeling properties

The COTS assembly must implement an additional requirement, expressing the coordination between the door and filling gap operation for security reasons. This requirement states that after an open request, the filling-gap should always deploy before the door is open:

$P^{\text{pasm}}: \text{always req\_open} \rightarrow \text{ack\_deploy before ack\_open}$

This safety property is modeled as an observer $M^{\text{pasm}}$ illustrated in Fig. 20.

4.4. Automatic error detection

The COTS assembly modeled above is verified using model checking. The verification shows that the assembly model does not satisfy $P^{\text{pasm}}$ and the model checking tool provides a counterexample which highlights the fact that the inputs req_open, req_close, req_deploy, req_withdraw, cmd_open, and cmd_deploy are responsible of the property's violation. Since $P^{\text{pasm}}$ is a safety property, then the automatic error correction using EDCS at step 3 is applied.

4.5. Automatic error correction

By applying the decision procedure presented in Section 3.3 the designer designates the set of req_open, req_close, req_deploy, req_withdraw as controllable inputs.

As property $P^{\text{pasm}}$ seems to concern only the Door and Filling gap COTSs, a first attempt is made to enforce it by EDCS on the model $M^p \parallel_{\text{IoC}} M^\text{f}$. The resulting controlled structure is shown in Fig. 21, in this figure, the solid-style input signals of the controller represent the $X$, whereas the dashed-style input signals represent $X_{\text{uic}}$.

4.6. Formal verification of the controlled system

The validation of the controller consists of formally verifying the guarantees $G^d$, $G^g$ and $G^f$. They must still hold after the addition of the controller. Besides, “event inventions” should never occur, and the controller should not delay forever input transactions.

The absence of event invention is checked by the properties: never $(\text{req\_}\neg\neg) \rightarrow (\text{req\_}\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\nel
Controlbuild is able to handle several kinds of models and translate them into the same PLC/Open pivot representation. In this project, only the use of state/transition models is advocated, graphically expressed as GRAFCET models, and dataflow models featuring logical gates and state variables.

Design models are automatically translated into synthesizable VHDL, via the hdlgen tool, which is a part of Controlbuild. At the end of this work, all this tool-chain has become unavailable to us, due to licensing issues. It has been replaced by other free of access tools for modeling and simulation: Xilinx StateCAD can model graphically synchronous state based designs, simulate them, and translate them into synthesizable VHDL.

The toolchain starts from synthesizable VHDL designs. It is possible to read and translate them into several tool-specific formats: the z/3z, SMV and nuSMV formats all have in common an input/state variable/transition function/output representation.

The first step is achieved by the Design Compiler (DC) tool, referred to as “RTL compiler” in Fig. 23. DC is a commercial tool provided by Synopsys. This compilation step, also known as RTL synthesis, performs the transformation of a model written into synthesizable VHDL, possibly containing high-level programming mechanisms, into a set of state variable and their corresponding transition functions. Everything is converted into the Boolean representation.

This result is further compiled and translated into the tool specific formats, in order to be able to feed the same design to either DCS or formal verification.

The DCS and the formal verification are achieved using academic free tools: Sigali for DCS, Cadence SMV (or alternatively, NUSMV) for formal verification.

The synthesized controller is assembled manually to the original COTS, once translated into synthesizable VHDL. The controller can also be re-translated into ControlBuild through the cont2comp tool (locally developed) for global simulation. This step has not been tested. All simulations were performed on VHDL models.

In the current implementation state, the functional requirements for either formal verification or DCS are manually handled by the designer, at the time of verification or synthesis.
6. Conclusion

This paper has presented a safe design method for COTS-based hardware embedded systems involving 5 steps starting from the COTS modeling and ending by simulating the final design which is ready to be installed on an FPGA chip. The method uses in synergy the Discrete Controller Synthesis and Formal Verification techniques in order to produce correct by construction COTS-based systems which can be kept in a COTS library in order to be reused. Specific issues related to the use of DCS in the hardware design context have been identified and addressed: (1) the structural compatibility between the controller and the system to control, (2) the integration of environment assumptions in solving the DCS problem and filtering the values coming from the environment, by the controller, (3) and the behavioral validation of the controller. An EDCS algorithm is proposed which calculates a supervisor for the system, while taking into account the COTS’ environment assumptions. The method proposed also includes compositional reasoning of the COTS’ environment assumptions and guarantees, which is out of the scope of this paper. The method has been illustrated over an industrial system: a control-command system of a train.

The addition of DCS into an embedded design flow constitutes the main originality of this contribution, compared to ordinary design methods which rely heavily on simulation and (sometimes) on formal verification. DCS appears to be useful in handling several aspects:

- automatic generation of proper coordination between COTS;
- design error masking: DCS can be used as a “patch” generation technique for COTS having design errors which cannot be modified;
- robustness with respect to hidden bugs: a COTS is not a perfect component, it can have undocumented bugs. Yet, DCS builds a controller on top of the code of the COTS assembly. If the target requirement can be ensured, a controller is generated which ensures it.

It has also been shown that there exists a real synergy between DCS and formal verification.

Another important and interesting specificity of this contribution is that it applies DCS without altering the COTS interfaces that are controlled.

Future directions of this work aim at using DCS for interface generation, as well as handling liveness requirements enforcement. This aspect was not covered by a straightforward application of DCS. It would be interesting to investigate the use of liveness counter-examples as hints for DCS. Another important direction is the compositional reasoning in the application of DCS to COTS-based design.

Acknowledgment

Special thanks to Bombardier Transport for providing the passengers’ access system’s specification.

References


